DTIC FILE COPY

NASA Contractor Report 182081
ICASE Report No. 90–53

AD-A227 105

ICASE

PARALLELIZATION OF IMPLICIT FINITE DIFFERENCE SCHEMES IN COMPUTATIONAL FLUID DYNAMICS

Naomi H. Decker
Vijay K. Naik
Michel Nicoules

Contract No. NAS1–18605
August 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

DTIC
ELECTE
OCT 03 1990
S E D

90 10 03 047

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Parallelization of Implicit Finite Difference Schemes in Computational Fluid Dynamics

Naomi H. Decker *
ICASE, NASA Langley Research Center
Hampton, VA 23665.

Vijay K. Naik
T. J. Watson Research Center
Yorktown Heights, NY 10598.

Michel Nicoules †
T. J. Watson Research Center
Yorktown Heights, NY 10598.

Abstract

Implicit finite difference schemes are often the preferred numerical schemes in computational fluid dynamics, requiring less stringent stability bounds than the explicit schemes. Each iteration in an implicit scheme, however, involves global data dependencies in the form of second and higher order recurrences. Efficient parallel implementations of such iterative methods, therefore, are considerably more difficult and non-intuitive. In this paper, we consider the parallelization of the implicit schemes that are used for solving the Euler and the thin layer Navier-Stokes equations and that require inversions of large linear systems in the form of block tri-diagonal and/or block penta-diagonal matrices. We focus our attention on three-dimensional cases and present schemes that minimize the total execution time. We describe partitioning and scheduling schemes for alleviating the effects of the global data dependencies. An analysis of the communication and the computation aspects of these methods is presented. The effect of the boundary conditions on the parallel schemes is also discussed. The ARC-3D code, developed at NASA Ames, is used as an example application. Performance of the proposed methods is verified on the Victor multiprocessor system which is a message passing architecture developed at the IBM, T. J. Watson Research Center.

*Research supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665-5225.

†Current address: Institut National Des Telecommunications, 91011, Evry, France.

1 Introduction

Efficient parallel implementation of any iterative method depends on the characteristics of both the particular application problem to be solved and the architecture of the parallel machine on which it is to be solved. Using iterative methods to find the numerical solution of a partial differential equation with boundary conditions generally involves more than just the repeated solutions of linear systems of the form

$$Ax = b. \tag{1}$$

The elements of A and b must also be evaluated at every iteration. Depending on the equations and the boundary conditions, these calculations can be a significant part of the solution cost. In many cases the data dependencies of the iterative method and those in evaluating the elements of A and b are considerably different. Therefore load balancing and communication requirements vary from one part of the iterative process to the next. Thus, evaluating the parallelism of an iterative method by considering just the cost of solving Equation 1 is not necessarily a good indicator of the parallelism of the corresponding iterative method for solving the partial differential equation.

In this paper we consider some of the issues involved in the efficient parallelization of iterative schemes in the overall context of a computational fluid dynamics (CFD) application. In particular, we examine a class of algorithms known as the implicit schemes for their parallelization properties on message passing, MIMD multiprocessor systems. We show here that, with appropriate rearrangement of computation and by using suitable partitioning strategy, one can extract high degree of parallelism from these schemes. The ARC-3D code, developed at NASA Ames, is used as a model application. Experimental results are presented from the implementations on the Victor multiprocessor system developed at IBM, T. J. Watson Research Center.

The organization of the rest of the paper is as follows. In the next section, various iterative methods, explicit and implicit, used in the CFD applications are surveyed and their important properties are discussed. The three dimensional Euler and the thin-layer Navier-Stokes equations for modeling the compressible flow of a gas over a solid body are described in Section 3. The numerical methods used for solving these model equations are considered in the remainder of the paper. The implicit schemes that are of interest are described in Section 4. In Section 5, these schemes are broadly classified into three categories and the parallel properties of each class is analyzed. The salient points of the ARC-3D code and the implementation aspects are described in Sections 6, 7 and 8. Results are presented in Section 9 and finally Section 10 concludes the paper.

2 Numerical Methods in CFD

Whether using finite-element, finite-difference, finite-volume or spectral methods, efficient methods for solving the time dependent Euler or Navier-Stokes equations typically involve two basic types of time stepping schemes: the explicit and the implicit. The most easily parallelizable and relatively inexpensive are the explicit schemes which can be performed simultaneously at every grid element, often with highly localized spatial data dependencies. However, these explicit methods, with a localized spatial domain of dependence, are characterized by a time step restriction imposed by the Courant-Friedrichs-Lewy (CFL) condition. The implicit schemes, on the other hand, are not restricted by the CFL condition and are often advantageous for stiff problems which contain several time scales or for solving steady state problems using the time dependent equations as a device for the iterative solution of the steady state equations. The time step of an implicit method may be

restricted by the need to maintain a desired level of accuracy or to maintain numerical stability. In general, the reduction in the number of time steps for an implicit method must be weighed against the increase in the cost of the implicit calculations. On uni-processor and vector architectures, the implicit schemes have proven to be superior in many instances. For multiprocessor systems, in addition to the above mentioned factors, the data dependencies must also be taken into account. Because of the inherent global spatial data dependencies that generally require the solution of large sparse linear systems, the implicit methods are less amenable to parallelization. In this paper we show that by suitable rearrangement of the computation steps and by using appropriate communication and data partitioning schemes, it is possible to retain the efficiency of the implicit schemes on parallel systems as well.

Examples of explicit time stepping schemes are the Lax Wendroff methods such as MacCormack's predictor/corrector scheme (MacCormack, [16],[17]), the linear multistep methods such as leap frog and Adams-Bashforth, and the one step multistage schemes such as Runge-Kutta (Jameson, Schmidt, Turkel, [12]). Note that, regardless of the explicit or implicit time stepping scheme used, many of the other calculations involved in typical CFD codes, such as evaluating the residuals or the pressure have data dependencies similar to the explicit schemes.

Implicit time stepping schemes appear in a variety different methods. These include the Approximate Factorization (AF) / Alternating Direction Implicit (ADI) methods (Brédif, [5]; Beam, Warming, [3]; Abarbanel, Dwoyer, Gottlieb, [1]) and the closely related Fractional Step methods (Yanenko,[29]) and the LU implicit methods (Jameson, Turkel, [13]). In recent years, various Jacobi and Gauss-Seidel type iterative methods have been applied to solve the flux-split equations, either using the Newton linearization, (Chakravarthy,[7]) or the switched evolution/relaxation (SER) method, (Mulder and Van Leer,[27]). Implicit methods are also used to precondition minimum residual or conjugate gradient algorithms, for example, the Incomplete LU (Meijerink, Van der Vorst, [18]) and the Strongly Implicit Procedure (Stone, [25]). Most of these methods have also been used as smoothers for multigrid methods or in conjunction with explicit schemes. (See [4], [9], [24], [26].) Finally, implicit calculations, in the form of residual averaging, are used in the FLO codes, (Jameson, [11]), to stabilize and accelerate the convergence of an explicit Runge-Kutta multigrid method.

3 Equations

We consider implicit finite difference schemes for the three dimensional Euler and thin-layer Navier-Stokes equations which model the compressible flow of a gas over a solid body. The domain in Cartesian coordinates, x, y and z is mapped onto a computational domain with a general curvilinear coordinate transformation given by

$$\tau = t,\ \xi = \xi(x,y,z,t),\ \eta = \eta(x,y,z,t),\ \zeta = \zeta(x,y,z,t).$$

The surface of the solid body is assumed to be on a plane given by ζ = constant. In these coordinates, the time-dependent thin-layer Navier-Stokes equations are

$$\frac{\partial Q}{\partial \tau} + \frac{\partial E}{\partial \xi} + \frac{\partial F}{\partial \eta} + \frac{\partial G}{\partial \zeta} = Re^{-1}\frac{\partial S}{\partial \zeta} \tag{2}$$

Accession For
GRA&I
TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

where Q is the vector of conserved quantities,

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{bmatrix}$$

and ρ, ρu, ρv, ρw and e are, respectively, the density, the x, y, and z Cartesian components of momentum and the total energy per unit volume. The vector fluxes are given by,

$$E = J^{-1} \begin{bmatrix} \rho U \\ \rho u U + \xi_x p \\ \rho v U + \xi_y p \\ \rho w U + \xi_z p \\ U(e+p) - \xi_t p \end{bmatrix}, \quad F = J^{-1} \begin{bmatrix} \rho V \\ \rho u V + \eta_x p \\ \rho v V + \eta_y p \\ \rho w V + \eta_z p \\ V(e+p) - \eta_t p \end{bmatrix}, \quad G = J^{-1} \begin{bmatrix} \rho W \\ \rho u W + \zeta_x p \\ \rho v W + \zeta_y p \\ \rho w W + \zeta_z p \\ W(e+p) - \zeta_t p \end{bmatrix}$$

with the contravariant velocities, U, V, and W, defined as:

$$\begin{aligned} U &= \xi_t + \xi_x u + \xi_y v + \xi_z w \\ V &= \eta_t + \eta_x u + \eta_y v + \eta_z w \\ W &= \zeta_t + \zeta_x u + \zeta_y v + \zeta_z w \end{aligned}$$

and J is the metric Jacobian.

For a perfect gas, the pressure, p, and speed of sound, a, are given by

$$\begin{aligned} p &= (\gamma - 1)(e - \frac{1}{2}\rho(u^2 + v^2 + w^2)) \\ a^2 &= \frac{\gamma p}{\rho} \end{aligned}$$

where γ is the ratio of specific heats.

The thin-layer viscous term on the right hand side of the equation is given by

$$S = J^{-1} \begin{bmatrix} 0 \\ \mu m_1 u_\zeta + (\mu/3) m_2 \zeta_x \\ \mu m_1 v_\zeta + (\mu/3) m_2 \zeta_y \\ \mu m_1 w_\zeta + (\mu/3) m_2 \zeta_z \\ \mu m_1 m_3 + (\mu/3) m_2 (\zeta_x u + \zeta_y v + \zeta_z w) \end{bmatrix}$$

where

$$\begin{aligned} m_1 &= \zeta_x^2 + \zeta_y^2 + \zeta_z^2 \\ m_2 &= \zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta \\ m_3 &= (u^2 + v^2 + w^2)/2 + P_r^{-1}(\gamma - 1)^{-1}(a^2)_\zeta. \end{aligned}$$

The parameters μ, Re and Pr are the dynamic viscosity, Reynolds number and Prandtl number, respectively.

The metric terms can be obtained by chain rule from the definitions of ξ, η and ζ. The curvilinear derivatives in terms of the Cartesian derivatives are given by:

$$\begin{aligned}
\xi_x &= J(y_\eta z_\zeta - y_\zeta z_\eta) & \xi_y &= J(z_\eta x_\zeta - z_\zeta x_\eta) & \xi_z &= J(x_\eta y_\zeta - x_\zeta y_\eta) \\
\eta_x &= J(z_\xi y_\zeta - z_\zeta y_\xi) & \eta_y &= J(x_\xi z_\zeta - x_\zeta z_\xi) & \eta_z &= J(y_\xi x_\zeta - y_\zeta x_\xi) \\
\zeta_x &= J(y_\xi z_\eta - y_\eta z_\xi) & \zeta_y &= J(z_\xi x_\eta - z_\eta x_\xi) & \zeta_z &= J(x_\xi y_\eta - x_\eta y_\xi)
\end{aligned}$$

$$\begin{aligned}
\xi_t &= -x_\tau \xi_x - y_\tau \xi_y - z_\tau \xi_z \\
\eta_t &= -x_\tau \eta_x - y_\tau \eta_y - z_\tau \eta_z \\
\zeta_t &= -x_\tau \zeta_x - y_\tau \zeta_y - z_\tau \zeta_z
\end{aligned}$$

and J is given by,

$$J^{-1} = x_\xi y_\eta z_\zeta + x_\zeta y_\xi z_\eta + x_\eta y_\zeta z_\xi - x_\xi y_\zeta z_\eta - x_\eta y_\xi z_\zeta - x_\zeta y_\eta z_\xi.$$

The equations presented above describe Navier-Stokes equations in three dimensions after applying the thin-layer approximation theory. The Euler equations, for inviscid flow, are obtained by setting the viscous term on the right hand side of Equation 2 to zero.

4 Implicit Methods

A numerical solution of Equation 2 can be obtained by discretizing in both time and space. We first describe the time discretizations which lead to the implicit schemes. A single step time discretization of Equation 2 can be written as

$$\begin{aligned}
\frac{Q^{n+1} - Q^n}{t^{n+1} - t^n} &+ \alpha\left(\frac{\partial E^{n+1}}{\partial \xi} + \frac{\partial F^{n+1}}{\partial \eta} + \frac{\partial G^{n+1}}{\partial \zeta} - Re^{-1}\frac{\partial S^{n+1}}{\partial \zeta}\right) \\
&= -(1-\alpha)\left(\frac{\partial E^n}{\partial \xi} + \frac{\partial F^n}{\partial \eta} + \frac{\partial G^n}{\partial \zeta} - Re^{-1}\frac{\partial S^n}{\partial \zeta}\right).
\end{aligned}$$

Here, Q^n is the state of the system at time step t^n, and, for example,

$$E^n = E(Q^n).$$

A Taylor's series expansion of the vector flux terms and the thin layer viscous flux term can be used to linearize the $(n+1)$ terms giving

$$\begin{aligned}
E^{n+1} &\approx E^n + A^n(Q^{n+1} - Q^n) \\
F^{n+1} &\approx F^n + B^n(Q^{n+1} - Q^n) \\
G^{n+1} &\approx G^n + C^n(Q^{n+1} - Q^n) \\
S^{n+1} &\approx S^n + M^n(Q^{n+1} - Q^n).
\end{aligned}$$

A, B, C and M, the 5×5 matrices, are the flux Jacobians, given by $\frac{\partial E}{\partial Q}$, $\frac{\partial F}{\partial Q}$, $\frac{\partial G}{\partial Q}$, and $\frac{\partial S}{\partial Q}$, respectively, and $\Delta Q^n = Q^{n+1} - Q^n$. These linearized implicit equations can be written in the "delta form" as:

$$\left[I + \alpha\Delta t\left(\frac{\partial}{\partial \xi}A^n + \frac{\partial}{\partial \eta}B^n + \frac{\partial}{\partial \zeta}C^n - Re^{-1}\frac{\partial}{\partial \zeta}M^n\right)\right]\Delta Q \tag{3}$$

$$= -\Delta t(\frac{\partial E^n}{\partial \xi} + \frac{\partial F^n}{\partial \eta} + \frac{\partial G^n}{\partial \zeta} - Re^{-1}\frac{\partial S^n}{\partial \zeta}).$$

The choice of the parameter α in the above equations determines the implicit scheme. When α is equal to one, this is a first-order backward Euler scheme, $\alpha = 1/2$ is a second-order Crank-Nicolson, or trapezoidal scheme, and $\alpha = 0$ is forward Euler, an explicit scheme. In the limit, as Δt approaches infinity, and if $\alpha = 1$, this becomes a linearized Newton's method.

We write Equation 3 more simply as:

$$(I + \alpha \Delta t N)\Delta Q = -\Delta t R \tag{4}$$

where

$$N = \frac{\partial}{\partial \xi}A^n + \frac{\partial}{\partial \eta}B^n + \frac{\partial}{\partial \zeta}C^n - Re^{-1}\frac{\partial}{\partial \zeta}M^n$$

and the explicit terms on the right hand side are given by

$$R = \frac{\partial E^n}{\partial \xi} + \frac{\partial F^n}{\partial \eta} + \frac{\partial G^n}{\partial \zeta} - Re^{-1}\frac{\partial S^n}{\partial \zeta}.$$

The space discretization is performed on a uniform grid in the generalized coordinate system. Using second order central space differencing of the five coupled equations with lexicographic ordering of the grid points, Equation 4 results in a large (5×5) block banded linear system. For three dimensional problems, the bandwidth is proportional to the number of grid points in a plane. Therefore, the direct solution of the above system is impractical.

Without changing the accuracy of the scheme, the bandwidth of the linear system can be reduced by factoring $I + \alpha \Delta t N$. The factored system can then be solved with a direct inversion of each of the factors. For example, in the Approximate Factorization (AF) methods, such as the Beam-Warming scheme, the linear operator is written as the product of three factors, one for each coordinate direction. Thus, $I + \alpha \Delta t N$ is factored as

$$(I + \alpha \Delta t \frac{\partial}{\partial \xi}A^n)(I + \alpha \Delta t \frac{\partial}{\partial \eta}B^n)(I + \alpha \Delta t \frac{\partial}{\partial \zeta}C^n - \alpha \Delta t Re^{-1}\frac{\partial}{\partial \zeta}M^n)\Delta Q^n = -\Delta t R^n. \tag{5}$$

Central finite differencing of each factor, yields a block tri-diagonal matrix, where the off-diagonal 5×5 blocks are dense. An alternative to the three way factorization is an LU factorization. Here, $I + \alpha \Delta t N$ is factored so that Equation 4 becomes:

$$LU\Delta Q^n = -\Delta t R$$

where L contains only backward-differenced and U contains only forward-differenced discretizations of the spatial derivatives. Ordering the grid points accordingly, L and U become lower and upper triangular matrices. The bandwidth of these matrices is as large as that of the unfactored system, but now the inversion can be accomplished in just two sequential sweeps over the grid points.

Iterative methods can also be used to solve the unfactored system. Various relaxation or splitting methods can be obtained by approximating data at off-diagonals by old data (at time t^n). Jacobi, Gauss-Seidel and SSOR methods can be used, in their many three dimensional forms. For example, the simplest is a point Jacobi method, where values at each point are updated using old values at all surrounding grid points. A more elaborate method is a red/black(checkerboard)-line Gauss-Seidel, obtained by performing tridiagonal solves on all 'red' lines in a plane, using the old data at the

'black' points, followed by tridiagonal solves on all black lines, using the new values at the red points. This could be done first in the $\xi - \eta$ plane, then in the $\eta - \zeta$ plane, and then in the $\zeta - \xi$ plane.

No matter which method, direct or iterative, it is the data dependencies which determine the available parallelism. In the next section, all of these methods are grouped into three distinct data dependence categories.

5 Parallelization of Implicit Methods

The schemes described above, although all are implicit, exhibit different degrees of parallelism. In general, the extent to which any of these methods is parallelizable can often be predicted by examining the nature of the inter-grid data dependencies in going from one time step to the next. We find it convenient to classify the methods into three categories based on the dependencies: the *parallel-by-point* schemes, the *parallel-by-line* schemes, and the *parallel-by-plane* schemes. These categories in effect characterize the granularity of the available parallelism and, quantitatively, the degree of the extractable parallelism. As a consequence, the implementation techniques for all the schemes within a category and their expected performance are similar.

Parallel-by-point	Parallel-by-line	Parallel-by-plane
Multistage methods (e.g., Runge-Kutta)	ADI/AF methods (e.g., Beam-Warming)	Implicit LU
Lax-Wendroff methods (e.g., MacCormack)	Fractional step methods	ILU and SIP
Multistep methods (e.g., leap frog)	Red/black-line Gauss-Seidel	Point/line Gauss-Seidel
Point Jacobi	Line Jacobi	SSOR

Figure 1: A classification of implicit methods based on available parallelism.

In the following, we describe the main properties of the three categories using a three-dimensional $n \times n \times n$ grid as an example. Note that the properties are unaffected even if the grid is non-uniform and even if it has unequal number of grid points in each of the three dimensions.

The 'parallel-by-point' schemes are the simplest. To advance to the next time step, computations can be done at each grid point independently of the computations at the other grid points. Depending on the discretization stencil used, values from the previous time step at one or more neighboring grid points are needed in the computations at any grid point. The time-explicit schemes are all examples of 'parallel-by-point' schemes. A few of the time-implicit schemes also fit in this category, for example, point Jacobi on the unfactored equations. The 'parallel-by-point' schemes are extensively analyzed in the literature for partitioning and for minimizing the communication overhead. (See, for example, Reed et al., [23].) On an $n \times n \times n$ grid, up to n^3 proces-

sors can be used without introducing any computational sequentiality. Any processor assignment scheme which allocates approximately the same number of points to each processor gives a good first approximation to balancing the computation load. On distributed memory architectures, a static three dimensional block partitioning of the grid is usually sufficient to minimize the communication overhead, provided the interprocessor communication network provides at least a three dimensional connectivity. In practice, two dimensional partitioning schemes also suffice in reducing the communication costs close to a minimum. Improved load balancing of the computation and minimization of the communication overhead is usually achieved by an analysis of the effect of the data dependencies at the boundaries.

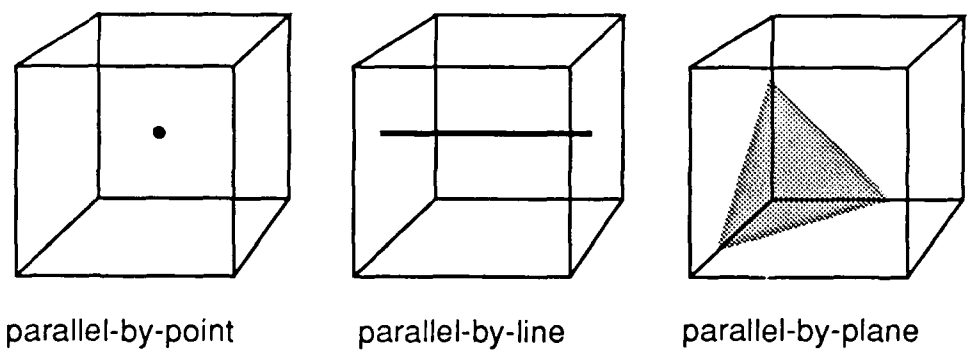

Figure 2: Types of parallelism.

'Parallel-by-line' schemes are a little more difficult. To advance to the next time step, the computation at each grid point is coupled to other grid points in a fixed coordinate direction, but can be done independently of the computations at grid points on all other lines in that coordinate direction. In other words, in the 'parallel-by-line' schemes the dependencies are such that all the grid points on a line must be updated together. The computations on different lines, however, may proceed without any dependence or communication delays. The quasi-one-dimensional algorithms such as ADI, AF and fractional step algorithms are in this category, as well as some of the line relaxation methods listed in Figure 1. For an $n \times n \times n$ grid, there are n^2 grid lines in each of the three dimensions and hence, as long as there are n^2 or less processors, the optimal sequential algorithms can usually be modified to give a good parallel algorithm. With more than n^2 processors, the solution process must usually be 'blocked out'. That is, each processor performs as many independent (parallel) computations on local data as possible before combining information with other processors. Having reduced the number of coupled equations, the remaining equations are solved by the same method by a subset of the processors. If more than one line is to be solved simultaneously, the rest of the processor continue the reduction on other lines. A typical example of this is when performing tridiagonal line solves. The sequentially optimal Thomas algorithm can be replaced by a substructuring/cyclic reduction algorithm to extract more parallelism. Note that

the cyclic reduction algorithms require more than twice as many floating point operations per grid point as the Thomas algorithm. Moreover, for these algorithms the dependencies fan out requiring richer interprocessor connectivity to reduce the communication overhead. The cyclic reduction type algorithms, however, have the advantage that more of these operations can be done in parallel (Heller, [8]; Ho and Johnsson, [10]).

The third category of 'parallel-by-plane' involves dependencies that span two or more dimensions and are more difficult to handle than the other two cases. LU factorization and Gauss-Seidel methods are examples of these methods. Extracting parallelism from LU factorization and Gauss-Seidel methods is possible, but is much more difficult, see (Naik,[19]). For a three dimensional problem, these methods are vectorizable along $j + k + l =$ constant planes, but the vector length varies. Similarly, the parallelization could be done in planes, at the cost of inefficient processor utilization near the corners of the computational domain. Because of the nonlinearity of the equations, the triangular LU factors are different at every point and at every time step. The parallelization methods of Anderson and Saad [2], which involve preprocessing overhead which must be amortized over many iterations to be worthwhile, are not useful in this context. Because of their improved convergence properties and increased stability, these methods warrant further investigation.

For the remainder of this paper, we consider the parallelization of a 'parallel-by-line' scheme: the Beam-Warming Approximate Factorization scheme. The AF methods have been extensively studied. They can be made to be total variation diminishing (TVD), (Chakravarthy et al.,[6]) and the boundary conditions are well understood, (LeVeque,[15]). The ARC-3D code developed at NASA Ames is based on this scheme. In the following , we first briefly described the key aspects of ARC-3D, referring the reader for details to Pulliam and Steger, [22], and then discuss possible parallel implementations. In Section 9 we describe our implementation on the Victor multiprocessor system at IBM, Yorktown Heights.

6 ARC-3D

A widely distributed, general purpose implicit finite difference code is the ARC-3D code developed at NASA Ames, (Pulliam and Steger, [21]). It is based on the Beam-Warming Approximate Factorization scheme, described in Section 4. The numerical solution of the factored equation involves inverting the three factors shown in Equation 5. Since each of the three factors represents a recursive coupling of information along a particular spatial direction, the inversion of a factor has global data dependencies, usually in the form of block tridiagonal matrices, in that direction.

Although the computation costs associated with Approximate Factorization scheme are considerably less than direct solves, they are still expensive. A significant reduction (40%) in computation can be achieved using the diagonalization of the blocks in the implicit operator, as developed by Pulliam and Chaussee [20]. The flux Jacobians, $\hat{A}^n$, $\hat{B}^n$ and $\hat{C}^n$ in Equation 5 can be diagonalized as:

$$\Lambda_\xi = T_\xi^{-1}\hat{A}^n T_\xi; \quad \Lambda_\eta = T_\eta^{-1}\hat{B}^n T_\eta; \quad \Lambda_\zeta = T_\zeta^{-1}\hat{C}^n T_\zeta.$$

For steady state calculations, a modified factored algorithm for the Euler equations, which is now at most first order accurate in time, is given by

$$T_\xi(I + \Delta t\partial_\xi\Lambda_\xi)\hat{N}(I + \Delta t\partial_\eta\Lambda_\eta)\hat{P}(I + \Delta t\partial_\zeta\Lambda_\zeta)T_\zeta^{-1}\Delta Q^n = \hat{R}^n$$

with $\hat{N} = T_\xi^{-1}T_\eta$ and $\hat{P} = T_\eta^{-1}T_\zeta$. For the explicit form of the Λ's, T's, N and P, see (Pulliam and Steger,[22]). A fourth order finite difference discretization of the implicit factors yields scalar penta-

diagonal and (5×5) block diagonal factors. Thus, the numerical solution using the diagonalization technique consists of inverting seven factors, four of which are in (5×5) block diagonal form and the other three, which correspond to the three spatial directions, are in scalar penta-diagonal form. As in the block tri-diagonal case, the inversion of the scalar penta-diagonal matrices has global dependencies in a particular spatial direction, but now the five variables at each grid point are decoupled.

We consider here, two versions of ARC-3D code for examining the implementation and performance issues in parallelizing some of the implicit schemes described and characterized in Sections 4 and 5. The first version corresponds to the standard way of computing the approximate factors and is based on the solution of the block tri-diagonal matrices. We refer to this version of ARC-3D as the *block tri-diagonal* version. The other version considered here is based on the diagonalization technique described above. This version of ARC-3D is referred to as the *scalar penta-diagonal* version. In the following, we describe the computation steps and the corresponding data dependencies in the two versions.

	penta-diagonal				block tridiagonal			
	$+$	$\times$	$\div$	$\sqrt{}$	$+$	$\times$	$\div$	$\sqrt{}$
RHS	290	356	37	3	290	356	37	3
IMPLICIT								
- setup	235	410	26	6	389	512	18	0
- forward solves	66	99	6	0	495	510	117	0
- back solves	30	30	0	0	75	75	0	0
TOTAL	331	539	32	6	959	1097	135	0

Table 1: Floating point operations per grid point, non-viscous case.

The implementation of the Beam-Warming scheme in generalized coordinates involves a sequence of separate tasks. These tasks include an initialization part where the computations corresponding to initial setup are made. This is followed by computations over each time step. Each time step requires the solution of Equation 5 to find the correction, ΔQ^n. The computation at each time step is accomplished numerically by solving either a (5×5) block tridiagonal system or by solving a scalar penta-diagonal system for each line, in each of the three spatial directions. Thus, for an $n \times n \times n$ grid, there are n^2 such systems to be solved for each direction. At the computational level, however, four separate tasks are necessary for advancing a time step. First, the discrete boundary conditions must be calculated at all boundaries. The inflow, outflow and solid body boundary conditions must be set, and any symmetry or singular conditions related to the computational domain must be enforced. Then the fluxes, numerical viscosity and the thin layer

viscous terms on the right hand side of Equation 5 are evaluated, using the appropriate difference formulas. Next, in the block tri-diagonal version, each of the three 5×5 coefficient matrices must be evaluated at every grid point in the domain and then and only then can each factor be inverted. For the scalar penta-diagonal version, each of the five scalar coefficients of the penta-diagonal system must be evaluated at every grid point for each variable before the penta-diagonal factor can be inverted. Finally, the flow variables, Q, must be updated:

$$Q^{n+1} = Q^n + \Delta Q^n.$$

We refer to these four tasks as BC, RHS, IMPLICIT and UPDATE, respectively.

In general, a typical application involves thousands of time steps. This amortizes the setup cost over the many iterations and hence the initialization cost is insignificant. The work involved in UPDATE is no more than five independent floating point additions at each grid point and hence is also negligible. So we restrict out attention to the three major tasks, BC, RHS and IMPLICIT.

The data dependencies of each of these three operations are quite different. In BC, only the subdomain consisting of boundary points are updated using values from only those interior points which are close to the boundary points. The pressure on the solid body is determined via the normal momentum equation, and because central differences of tangential derivatives of the pressure are used, scalar tridiagonal inversions must be performed along the solid body. RHS is explicit and is therefore completely data parallel, but some of the calculations must be done at all grid points with the final computation at only the interior points. Finally, IMPLICIT involves either the inversion of (5×5) block tri-diagonal systems or scalar penta-diagonal systems in each of the three coordinate directions.

The number of floating point operations per grid point for RHS and IMPLICIT in the non-viscous case are shown in Table 1 for the two versions of ARC-3D. The number of floating point operations per grid point in BC are relatively small compared to those in RHS and IMPLICIT and for that reason they are not shown. Clearly, for both versions of ARC-3D, the cost of IMPLICIT is the dominant cost.

7 Algorithms

To bring out the salient points for parallel implementations, we now describe the three main tasks of ARC-3D at the algorithmic level. First, the algorithmic aspects of IMPLICIT are presented in some detail and then those for RHS and BC are briefly described. The task of IMPLICIT consists of solving linear systems by inverting either block tri-diagonal or scalar penta-diagonal matrices. In the block tri-diagonal case, for each direction as many systems need to be solved as there are lines of grid points in that direction. In the scalar penta-diagonal version, for each variable and for each direction, the number of systems solved is equal to the number of lines of grid points in that direction. In addition to inverting scalar penta-diagonal matrices, this version requires inversion of block diagonal matrices. However, the inverse of each block is known analytically, and the block diagonal inversions can be done without requiring information from any other grid point. The algorithms used for solving block tri-diagonals and scalar penta-diagonals are similar to those used for inverting scalar tri-diagonals, which we describe next.

Sequentially, the most efficient algorithm for inverting a tri-diagonal system is the Thomas algorithm which is a special case of the Gaussian elimination. To solve a system of size n, with

coefficients A, B and C and right hand side f, given by,

$$A_j u_{j-1} + B_j u_j + C_j u_{j+1} = f_j$$

where u_0 and u_{n+1} are known, the Thomas algorithm performs two sweeps over the unknowns u_1 through u_n. In the forward sweep, coefficients P_{j+1} and q_{j+1}, $j = 1, \cdots, n$, are computed, where

$$\begin{aligned} P_{j+1} &= (B_j - A_j P_j)^{-1} C_j \\ q_{j+1} &= (B_j - A_j P_j)^{-1} (f_j - A_j q_j). \end{aligned}$$

The unknowns u_j, $j = n, \cdots, 1$, are determined in the back sweep from

$$u_j = q_{j+1} - P_{j+1} u_{j+1}.$$

Note that the computation requires storage of all P_{j+1} and q_{j+1} from the forward sweep for computing the unknowns u_j in the back sweep. Clearly, both phases of the Thomas algorithm have recursive data dependencies and hence the algorithm, when used for solving a single system, is sequential. However, when several independent such systems need to be solved as in the case of IMPLICIT, the computations of these systems can be pipelined. Thus, the above algorithm can be parallelized by assigning to each processor parts of computations from several independent tri-diagonal system solves. In such a scheme, if a processor is to compute unknowns u_{j_p} through u_{k_p} from one of the systems, then it receives the coefficients P_{j_p} and q_{j_p} from a preceding processor and then proceeds to compute the coefficients through P_{k_p} and q_{k_p} and sends the final coefficients P_{k_p} and q_{k_p} to the succeeding processor. The same is repeated for all other systems assigned to it. After completing all the forward sweeps, the back sweeps are performed. First the values of u_{k_p+1} are received for each system from the succeeding processor and back sweep is completed by computing the unknowns u_{k_p} through u_{j_p}. The computed value of u_{j_p} is sent to the preceding processor and the process is repeated for all the systems assigned to that processor. This we refer to as the *pipelined version* of the Thomas algorithm. Note that all the coefficients from all the forward sweeps must be stored until the back sweeps are completed.

The tri-diagonals can also be computed using the substructuring or the cyclic reduction algorithms (See, e.g., Ho and Johnsson [10]). However, these schemes are computationally expensive. The cost of Thomas algorithm for scalar tri-diagonals is approximately 8 floating point operations, whereas the cyclic reduction type algorithms require about 17 floating point operations per variable. The Thomas algorithm is a sequential algorithm and if only one tri-diagonal system is to be solved, then the available of parallelism is negligible. The cyclic reduction type algorithms, on the other hand, are highly parallelizable. In the next section we describe partitioning schemes for extracting parallelism from the Thomas algorithm in the context of ARC-3D and show that the available parallelism is sufficient for the machine model assumed here.

For the block tri-diagonal case, each of the coefficients A and C is a dense 5×5 block, matrix B is a 5×5 diagonal block, and u_j and f_j are vectors of length 5. The computed coefficients P_j and q_j are, respectively, dense 5×5 matrix and 5×1 vector. Thus, in addition to the cost of computing the coefficients, the corresponding Thomas algorithm involves factoring a 5×5 matrix and computing six matrix-vector products at each grid point. The scalar penta-diagonal case has five scalar coefficients instead of three for each variable at each grid point. The corresponding Thomas algorithm solves penta-diagonal systems in a similar fashion as the tri-diagonal system. Note that the dependencies are in the form of the second order recurrences. For each variable, the

forward sweep computes three coefficients and the computation of u_j in the back sweep involves values at u_{j+1} and u_{j+2}.

In RHS, the fourth order spatial discretization of the right hand sides leads to 13 point stencils at the interior points, modified to smaller stencils near the boundary. These calculations are performed at each grid point and they involve explicit data dependencies, i.e., they use known information. Therefore, the computations for RHS may be completed independently. The implementation is similar to Jacobi relaxation and the computations may be performed in any order.

In BC, the inflow/outflow boundary points can also be updated independently, but the values at boundary points along the solid body are coupled and must be updated using tridiagonal solves in both the ξ and η directions. We use the scalar Thomas algorithm for these tridiagonal solves.

8 Parallel Implementation

Efficient parallel implementation requires extraction of maximum parallelism with negligible communication overhead. This entails partitioning of the data across processors and rearranging the computations so that the load is evenly balanced and the communication overhead is kept to a minimum. In practice achieving both the goals simultaneously is difficult. Furthermore, since the data dependencies of BC, RHS, and IMPLICIT are not the same, reaching this goal is even harder. We describe here a class of partitioning schemes that are suitable for MIMD architectures where the number of processors is small compared to the number of grid points. For the following discussion assume that p processors are available and that the computational domain is an $l \times m \times n$ grid.

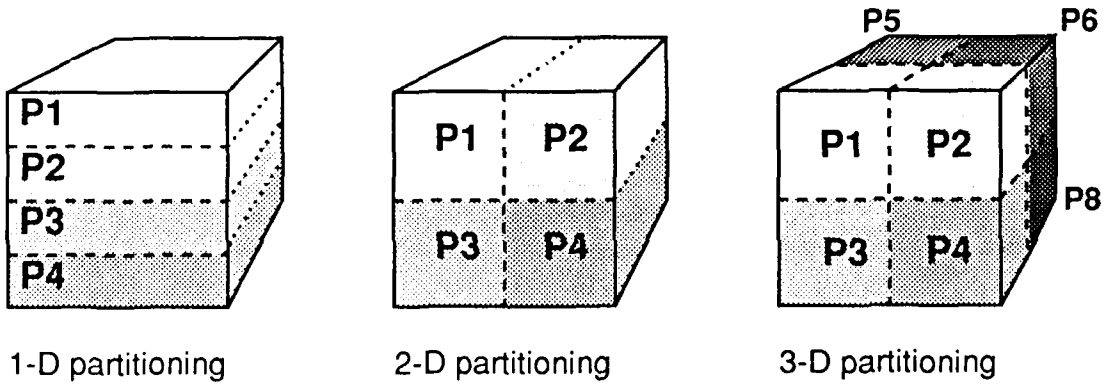

Figure 3: Uni-partition schemes.

8.1 Uni-partition Schemes

We refer to a partitioning scheme as a *uni-partition scheme* when the domain of computation is subdivided into p partitions and one partition is assigned to each processor. There are several ways of partitioning the domain in this manner. The simplest and the most commonly used are those where the domain is partitioned evenly, or almost evenly, along one, two, and three spatial

dimensions. If the computational domain has at least p grid points in one of the spatial directions then partitioning the grid into p slices along that direction gives the 1-D uni-partition scheme. If the number of grid points on a plane is at least equal to p, then the 2-D uni-partitioning scheme is possible. Here the grid is partitioned along two spatial directions. Similarly, in the 3-D uni-partitioning scheme, the domain is divided along all the three dimensions. See Figure 3.

The 3-D partitioning scheme has the smallest surface to volume ratio whereas the 1-D scheme has the largest. Thus, for RHS, where the data dependencies are local, the 3-D partitioning scheme reduces the volume of data communicated. However, in this scheme, some processors may have to communicate with at least six other processors and possibly with up to 12 other processors. This scheme is also unattractive if a three dimensional mesh cannot be mapped efficiently on the underlying inter-processor communication network. Furthermore, the use of 13 point stencil in RHS makes it necessary to maintain a buffer containing two rows or two columns worth of extra information from the neighboring partitions at each surface of the partition that is adjacent to another partition. This memory overhead increases as the number of grid points per processor decreases. In the extreme case where one grid point is assigned to a processor, the information at the surrounding 12 grid points may have to be buffered.

Sequential Delay	Memory Required
$O(n)$	$O(n^3/p)$
$O(n^2)$	$O(n^2/p)$
$O(n^3)$	$O(n/p)$

Figure 4: The pipelining delay and memory requirement tradeoff.

For the IMPLICIT part, the 1-D partitioning scheme requires inter-processor communication in only one of the three directions. Thus, for the 1-D partitioning scheme, the adverse effect of the data dependencies is restricted along only one direction. The effect of data dependencies may be further reduced by pipelining the forward and back sweeps of IMPLICIT, as observed in (Johnson, Saad and Schultz,[14]). The larger the number of lines that can be pipelined, the lesser the effect of sequential data dependencies present in the Thomas algorithm. If there is no pipelining, then each of the line solves is done sequentially, irrespective of the number of processors used. However, if all the line solves are pipelined, then the net effect is that the computation may be completed in time equal to one line solve performed sequentially plus the time required to compute the remaining line solves with perfect parallelism. This cost can be further reduced by starting the computations from both sides simultaneously. This reduces the sequential cost of filling the pipe by half. In the 2-D and 3-D partitioning schemes, the data dependencies of IMPLICIT are spread across processors along two and three dimensions, respectively. This gives rise to data dependency delays in two and

three directions. Thus, if no pipelining of computations is performed, then the 3-D partitioning scheme is the most adversely affected and has very little parallelism. With full pipelining, the 2-D and 3-D cases have a delay of two and three sequential line solves, respectively. Note from Table 1, that for the type of problems considered here, the cost of one sequential line solve can be very high and may quickly dominate the total parallel cost. This is particularly true when the number of grid points assigned to each processor is small. By assigning a sufficiently large number of grid points to each processor, one may reduce the effect of the sequential part in the pipelined computation. However, there is a penalty to be paid in terms of memory. In order to be able to pipeline the computations, it is necessary to store the intermediate results of all the forward solves that are pipelined. This entails a memory overhead of storing an extra 5×5 matrix at each grid point for the block tri-diagonal case. In the penta-diagonal case the memory overhead is eleven coefficients per grid point along each direction. This gives rise to an interesting tradeoff between the memory requirement and the benefit of pipelining the IMPLICIT computations. See Figure 4.

Under the uni-partitioning scheme, the substructuring or the cyclic reduction type algorithms may also be used in IMPLICIT. These algorithms have higher degree of available parallelism, but there is a penalty of almost doubling the computation cost. Thus, the 1-D and 2-D partitioning schemes with substructuring generally fair poorly as compared to the Thomas based IMPLICIT. For 3-D partitioning schemes, the communications is global in all the three dimensions, making these schemes communication intensive.

In Table 2, some of the issues described above are quantified for computing scalar tri-diagonals along all the three dimensions of an $n \times n \times n$ grid and using p processors, neglecting lower order terms. These complexities assume $p \leq n$, though the complexities for the 2-D Pipelined and 3-D Diagonal scheme remain valid for $p \leq n^2$ and the 3-D complexity is valid up to $p \leq n^3$. The multi-partition schemes are described in the next section. The pipelined methods use the Thomas algorithm with 1-D, 2-D and 3-D partitioning. The arithmetic operations count gives the total parallel computation cost, including the dependency delays. The column 'words copied' gives the amount of data that must copied into the memory because of the partitioning scheme used. The number of messages is the minimum number of messages that must be encountered in a sequence from the beginning to the end of the computation. This sequence of messages represents minimum communication delay in that partitioning scheme because of the message overhead. This number takes into account all the messages sent by all the processors in parallel. The message size is in terms of the number of words per message. For Thomas based algorithms, two words must be sent per line in the forward sweep and one word in the back sweep. The message size for the forward sweep could be increased to 2 words, reducing the number of messages sent in the forward sweep by a factor of 2. The substructure algorithm uses 1-D partitioning, with the Thomas algorithm in the two directions which need no communication and the substructure algorithm in the direction which crosses processor boundaries.

There are two major drawbacks of the uni-partitioning schemes described above. First, load balancing is difficult. In practice, the problems to be solved may have grid sizes that cannot be evenly divided among the processors. Thus, if the granularity of the load distribution is to be retained at grid level, then some partitions may end up with a higher number of grid points to compute than the others. Since the computations per grid point are significant, the effect of this load imbalance may be considerable. This is especially true when the number of processors is large and the number of grid points assigned to a processor is relatively small. For the partitioning schemes described above, the problem is compounded by the fact that the load imbalance may be of

	arithmetic op's	words copied	number of messages	message size
Uni-partition				
Pipelined:				
1-D	$24\frac{n^3}{p}(1+\frac{p}{3n^2})$	-	$3(n^2+p)$	1
2-D	$24\frac{n^3}{p}(1+\frac{2p}{3n^2})$	-	$6(1+\frac{p}{n^2})\frac{n^2}{\sqrt{p}}$	1
3-D	$24\frac{n^3}{p}(1+\frac{p}{n^2})$	-	$9(1+\frac{p}{n^2})\frac{n^2}{p^{2/3}}$	1
Substructure	$33\frac{n^3}{p}$	$20n^2$	$2\log p$	$10n^2$
Multi-partition				
Transpose	$24\frac{n^3}{p}$	$8\frac{n^3}{p}\log p$	$2\log p$	$2\frac{n^3}{p}$
Diagonal:				
2-D	$24\frac{n^3}{p}$	-	$6n^2$	1
3-D	$24\frac{n^3}{p}$	-	$9n^2$	1

Table 2: Complexities of scalar tridiagonals in three dimensions; n^3 points, p processors.

the order of lines of grid points or even planes of grid points. This adversely affects the performance irrespective of whether only a small number of processors or a large number of processors are used. If non-standard partitions are used, then the load may be better balanced, but the communication and programming costs may turn out to be unreasonable. The second drawback applies even when the grid can be evenly divided among the processors. This is because of the uneven distribution of the computational work at the grid points. The interior points which form the major bulk of the computation have higher amounts of computational work associated with them than the grid points on the boundary. With this unevenness, any grid-level partitioning scheme results in a load imbalance. Furthermore, the data dependencies of BC, RHS, and IMPLICIT are totally different and BC applies only to the boundary points, RHS is applied over all the grid points, and IMPLICIT is applied only to the interior points. This adds to the data dependence delays even if the computational work is perfectly divided among the processors, since the processors with

partitions that include the boundary grid points are required to work on BC and RHS while those with only the interior points are not. One way to reduce the detrimental effects of load imbalance is to assign multiple partitions to each processor. This is described in the next section.

8.2 Multi-partition Scheme

In a *multi-partition* scheme, the computational domain is divided into a number of sub-partitions that is larger than the number of processors and each processor is assigned more than one such sub-partition. An obvious advantage is that the granularity of each sub-partition may be made much smaller than that in the uni-partition scheme. This gives better control over the computational load distribution. Moreover, as we describe next, not only is it possible to better balance the overall computational work, but also the load may be balanced at the task level so that the additional data dependency delays are not introduced. In the following, we describe one such scheme that we refer to as the *diagonal partitioning* scheme. A partitioning scheme similar to the 2-D version presented below is described by Johnsson et al. for implementing ADI methods on multiprocessor systems [14].

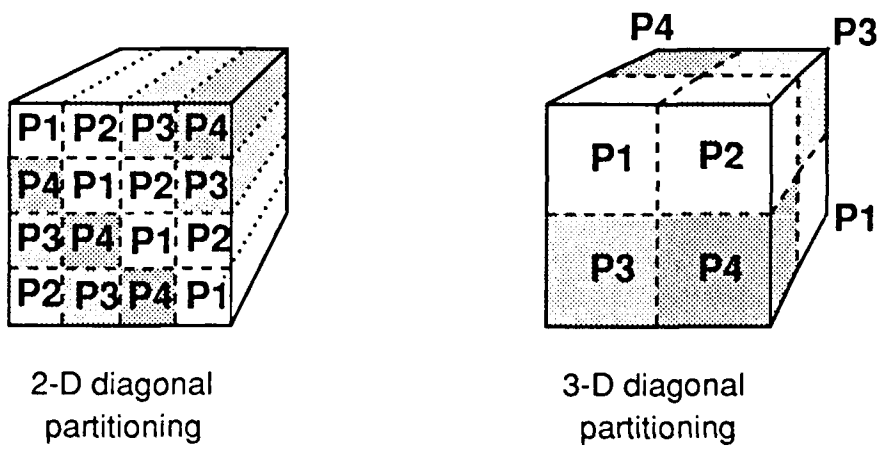

Figure 5: Multi-partition schemes.

In the diagonal partitioning scheme, the domain is partitioned in a manner similar to that of the 2-D or the 3-D uni-partitioning scheme described above. We refer to these as the *2-D diagonal* and *3-D diagonal* partitioning schemes, respectively. In the former case, the domain is subdivided into p^2 sub-partitions and in the later case it is subdivided into $p^{3/2}$ sub-partitions. In the 2-D diagonal partitioning scheme, each processor is assigned one sub-partition from each row and from each column of sub-partitions. Similarly in the 3-D diagonal partitioning scheme, each processor is assigned one sub-partition from each plane of sub-partitions. This arrangement results in assigning sub-partitions to a processor that form diagonal chains in two or three dimensions. Note that, for the 2-D diagonal partitioning scheme to be applicable, the computational grid should have at least p grid points on each side of one of the 2-D planes and for the 3-D diagonal partitioning case the computational grid should have at least $p^{1/2}$ grid points along each of the three dimensions.

The above described diagonal partitioning and assignment schemes require each processor to

compute all the three tasks: BC, RHS, and IMPLICIT. Because each processor is assigned one sub-partition from each row and column of sub-partitions, the computation is much better balanced even when the number of grid points along each dimension of the grid is not evenly divisible by p or $p^{1/2}$. An interesting aspect of the 2-D diagonal partitioning scheme is that any processor may need to communicate with only two other processors even though there are four sub-partitions surrounding each sub-partition. In the 3-D case, each processor needs to communicate with six other neighbors.

The diagonal schemes, however, have significantly larger surface area that is adjacent to other sub-partitions. This results in higher communication requirements. The communication and computation requirements in the scalar tridiagonal case for the 2-D diagonal partitioning scheme are shown in Table 2.

If the communication overhead is not substantial then the diagonal partitioning schemes have a clear advantage over the 2-D and 3-D uni-partitioning schemes described earlier. This can be seen by observing the maximum achievable speedups. For example, with the 3-D diagonal partitioning scheme applied to an $n \times n \times n$ grid and using n^2 processors, the maximum achievable speedup is n^2. The same with the 2-D uni-partitioning scheme is $3n^2/5$ and with 3-D uni-partitioning scheme it is $n^2/2$.

Another variation of multi-partitioning scheme is the case where the domain is partitioned into p^2 sub-partitions as in the case of 2-D uni-partitioning scheme, but the partitions are switched during the computation of IMPLICIT. Here, each processor is initially assigned p sub-partitions all from the same row of sub-partitions. After completing the IMPLICIT in two of the directions that lie in the plane of sub-partitions assigned to the processor, the processors transpose the sub-partitions before working on the final dimension of IMPLICIT. With this scheme, there is no communication during the IMPLICIT computation except for the transpose. The advantage is that there are no data dependency delays. The costs associated with this scheme for the scalar tri-diagonal case are shown in Table 2.

9 Implementation on Victor

We now describe some of the results from the implementations on the Victor multiprocessor system developed at T. J. Watson Research Center, Yorktown Heights. Victor is a message passing MIMD architecture with Inmos T800 transputer as the processing unit. Each node is associated with 4 MB DRAM and is connected to four other nodes forming a 2-dimensional grid interconnection. Victor is a modular architecture that can be configured from 16 to 256 nodes (Wilcke et. al.,[28]). All the implementations were made using the Inmos 3L parallel Fortran environment. The 3L-supported thread routines were used to mimic an asynchronous communication environment for overcoming some of the difficulties posed by the synchronous communication supported by the transputer hardware.

For all the results presented here, we use the flow past a semi-infinite hemisphere-cylinder body as a model problem. The computations were performed on a grid with 30 points in the axial direction, 12 points circumferentially and 30 points in the normal direction. Figures 6 and 7 show the body and the associated grid in the curvilinear coordinates. All the computations were performed with double precision arithmetic.

We have considered three different partitioning schemes for both the block tri-diagonal and the scalar penta-diagonal versions. These are the 1-D and 2-D uni-partitioning schemes, and the 2-D

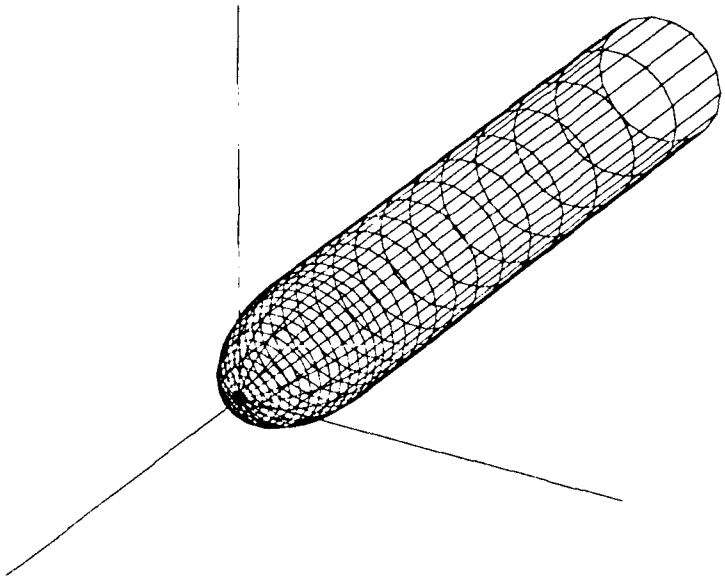

Figure 6: The hemisphere-cylinder body.

diagonal partitioning scheme. In all the cases the partitioning of the computational domain was along the axial and the normal directions only.

	(1,1)	(2,2)	(3,3)	(4,4)
BC	.988	.501	.385	.317
RHS	30.1	8.42	4.26	2.44
IMPLICIT	199.	51.4	26.4	13.2
TOTAL	230.	60.3	31.0	15.9

Table 3: Timings for the block tridiagonal version, using 1, 4, 9 and 16 processors (in seconds per time step).

The results from the implementations of the scalar penta-diagonal and block tri-diagonal versions of ARC-3D using the 2-D uni-partitioning schemes are compared in Tables 3 through 6. In all the cases the timings are given for each of the three tasks for four different processor configurations. These are a 1 processor (1,1), 4 processors (2,2), 9 processors (3,3), and 16 processors (4,4). In Tables 3 and 4 actual wall clock timings per time step are given. The speedups over one processor timings are given in Tables 5 and 6. It can seen that, though the block tridiagonal version is more expensive than the penta-diagonal version, it is more efficient in extracting the available parallelism.

An interesting observation is the comparison of the performance of 9 processors with that of

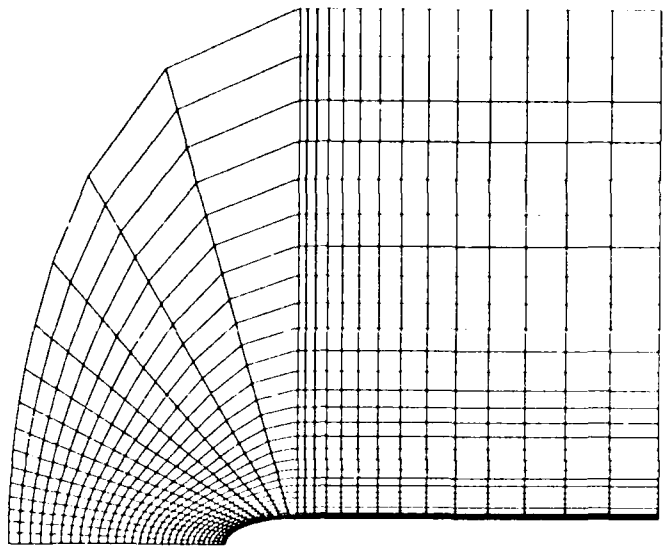

Figure 7: A cross-section of the grid in curvilinear coordinates.

	(1,1)	(2,2)	(3,3)	(4,4)
BC	.988	.560	.426	.438
RHS	30.2	8.52	4.36	2.57
IMPLICIT	64.6	17.1	8.93	4.49
TOTAL	95.8	26.2	13.7	7.50

Table 4: Timings for the penta-diagonal version, using 1, 4, 9 and 16 processors (in seconds per time step).

16 processors. For the problem size considered here and for the 2-D uni-partitioning schemes, the interior points are evenly divisible among the 16 processors, but are not divisible equally among 9 processors. The effect of this load imbalance is clear from the Tables 5 and 6.

The relative performance of the three tasks BC, RHS, and IMPLICIT are also to be noted. As expected, BC performs poorly, because most of the computation in BC is along the solid body which is computed by only 4 processors in the case of 16 processor implementation and by only 3 processors in the case of 9 processor implementation. RHS, although easily parallelizable, performs poorly as compared to IMPLICIT mainly because of the load imbalance. The partitions divide the interior points evenly, but this is not true when the boundary points are also included. In RHS, computations are performed at the interior as well as at the boundary points which affects its performance. IMPLICIT has only the interior points to compute and it performs very well.

Even here there is a slight load imbalance that is not obvious. The load imbalance is caused by the fact that the computations at the first point of a line solve are considerably less than those at the rest of the interior points. However, the sequential effect of the pipelining of IMPLICIT masks this imbalance.

The diagonal multi-partitioning schemes are much harder to implement than any of the uni-partition schemes. The difficulty arises in efficiently managing the multiple sub-partitions assigned to each processor. Since the number of sub-partitions assigned to a processor increases as the total number of processors working on the problem increases, the overhead for large number of processors can be very high, especially in the 2-D diagonal case where the relative gains are small.

	(1,1)	(2,2)	(3,3)	(4,4)
BC	1.00	1.97	2.57	3.12
RHS	1.00	3.57	7.07	12.3
IMPLICIT	1.00	3.87	7.54	15.1
TOTAL	1.00	3.81	7.42	14.5

Table 5: Speedups for the block tridiagonal version, using 1, 4, 9 and 16 processors.

Currently, we are in the process of completing this implementation and the results seen so far are encouraging. For the 2-D diagonal version, the assignment scheme results in each processor having to communicate with only two other processors. These neighboring processors remain the same throughout the entire computation. Thus, if the processors can be arranged in a loop, the 2-D diagonal version can be mapped onto these processors which results in nearest neighbor communication for BC, RHS, and IMPLICIT. Mapping a loop onto a mesh is relatively easy. However, to retain nearest neighbor communication only the loops with even number of processors are permissible.

	(1,1)	(2,2)	(3,3)	(4,4)
BC	1.00	1.76	2.32	2.26
RHS	1.00	3.54	6.93	11.8
IMPLICIT	1.00	3.78	7.23	14.4
TOTAL	1.00	3.66	6.99	12.8

Table 6: Speedups for the penta-diagonal version, using 1, 4, 9 and 16 processors.

10 Conclusions

We have attempted to show, by analysis and experimentation, the extent to which CFD applications based on implicit scheme can be parallelized. A specific example of an implicit scheme involving

line solves in all three coordinate directions was considered. It is observed that the affect of the various data dependencies of the different parts of the algorithm adversely affect the parallelism, but these can be minimized with an appropriate partitioning strategy.

Acknowledgment: The authors would like to thank Danny Shea for making available a 16 processor Victor at the Hawthorne site.

References

[1] S. Abarbanel, D. Dwoyer, and D. Gottlieb. Stable implicit finite-difference methods for three-dimensional hyperbolic systems. ICASE Report 82-39, 1982.

[2] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. Technical Report CRSD Rpt. No. 794, Center for Supercomputing Research and Development, University of Illinois, 1988.

[3] R. W. Beam and R. F. Warming. An implicit finite difference algorithm for hyperbolic systems in conservation form. *Journal of Computational Physics*, 23:87–110, 1976.

[4] M. Brédif. A fast finite element method for transonic potential flow calculations. AIAA paper 83-0507, 1983.

[5] M. Brédif. Finite element calculation of potential flow around wings. ONERA TP-1984-068, 1984.

[6] S. Chakravarthy, K-Y. Szema, U. Goldberg, and J. Gorski. Application of a new class of high accuracy TVD schemes to the Navier-Stokes equations. AIAA Paper 85-0165, AIAA 23rd Aerospace Sciences Meeting, Reno, 1985.

[7] S. R. Chakravarthy. Relaxation methods for unfactored implicit upwind schemes. AIAA Paper 84-0165, AIAA 22rd Aerospace Sciences Meeting, Reno, 1984.

[8] D. E. Heller. A survey of parallel algorithms in Numerical Linear Algebra. *SIAM Review*, 20:740–777, 1978.

[9] P. W. Hemker and S. P. Spekreijse. Multigrid solution of the steady Euler equations. In D. Braess, W. Hackbusch, and U. Trottenberg, editors, *Advances in Multi-Grid Methods*, Notes on Numerical Fluid Dynamics, Volume 11. Vieweg, 1985. Oberwolfach Meeting, December 1984.

[10] C. T. Ho and S. L. Johnsson. Optimizing tridiagonal solvers for alternating direction methods on boolean cube multiprocessors. *SIAM J. Sci. Stat. Compt.*, 11:563–592, 1990.

[11] A. Jameson. Solution of the Euler equations for two-dimensional transonic flow by a multigrid method. Technical Report MAE 1613, Princeton University, 1983.

[12] A. Jameson, W. Schmidt, and E. Turkel. Numerical solution of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes. AIAA Paper 81–1259, AIAA 14th Fluid and Plasma Dynamics Conference, Palo Alto, June 1981.

[13] A. Jameson and E. Turkel. Implicit schemes and LU decompositions. *Mathematics of Computation*, 37:385–397, October 1981.

[14] S. L. Johnsson, Y. Saad and M. Schultz. Alternating direction methods on multiprocessors. *SIAM J. Sci. Stat. Comput.*, 8:686–700, 1987.

[15] R. LeVeque. Intermediate boundary conditions for LOD, ADI, and approximate factorization methods. ICASE Report 85-21, 1985.

[16] R. W. MacCormack. The effect of viscosity on hypervelocity impact cratering. AIAA Paper 69-354, 1969.

[17] R. W. MacCormack. Current status of numerical solutions of the Navier-Stokes equations. AIAA Paper 85–0032, AIAA 23rd Aerospace Sciences Meeting, Reno, January 1985.

[18] J. A. Meijerink and H. A. Van der Vorst. An iterative solution method for linear problems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137), 1977.

[19] V. K. Naik. *On the Computation and Communication Tradeoffs and their impact on the performance of Asynchronous Multiprocessor Systems.* PhD thesis, Duke University, 1988.

[20] T. Pulliam and D. Chaussee. A diagonal form of an implicit approximate-factorization algorithm. *Journal of Computational Physics*, 39:347–363, 1981.

[21] T. Pulliam and J. Steger. Implicit finite-difference algorithm for hyperbolic systems in conservation law form. *AIAA Journal*, 18:159, 1980.

[22] T. H. Pulliam and J. L. Steger. Recent improvements in efficiency, accuracy, and convergence for implicit approximate factorization algorithms. AIAA Paper 85-0360, AIAA 23rd Aerospace Sciences Meeting, Reno, 1985.

[23] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. Comput.*, C-36:845–858, 1987.

[24] N. L. Sankar. A multi-grid strongly implicit procedure for two-dimensional transonic potential flow problems. AIAA paper 82-0391, 1982.

[25] H. L. Stone. Iterative solution of implicit approximations of multi-dimensional partial difference equations. *SIAM J. Numer. Anal.*, 5(3), 1968.

[26] A. J. van der Wees. *A Nonlinear Multigrid Method for Three-Dimensional Transonic Potential Flow.* PhD thesis, Technische Universiteit Delft, 1988.

[27] B. van Leer and W. Mulder. Relaxation methods for hyperbolic conservation laws. In F. Angrand, A. Dervieux, J. Desideri, and R. Glowinski, editors, *Numerical Methods for the Euler equations of fluid dynamics*, pages 312–333. SIAM, 1985.

[28] W. Wilcke, D. Shea, R. Booth, D. Brown, M. Giampapa, L. Huisman, G. Irwin, T. Ma, T. Murakami, F. Tong, P. Varker, D. Zukowski. The IBM Victor Multiprocessor Project. Proceedings of the 4th Int. Conference on Hypercubes, Apr. 1989.

[29] N. N. Yanenko. *The Method of Fractional Steps.* Springer-Verlag, New York, 1971.

Report Documentation Page

1. Report No. NASA CR-182081 ICASE Report No. 90-53	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle PARALLELIZATION OF IMPLICIT FINITE DIFFERENCE SCHEMES IN COMPUTATIONAL FLUID DYNAMICS		5. Report Date August 1990
		6. Performing Organization Code
7. Author(s) Naomi H. Decker, Vijay K. Naik, Michel Nicoules		8. Performing Organization Report No. 90-53
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18605
		13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration, Langley Research Center, Hampton, VA 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Richard W. Barnwell

Proceedings of Copper Mountain Conference on Iterative Methods, 1990

Final Report

16. Abstract

Implicit finite difference schemes are often the preferred numerical schemes in computational fluid dynamics, requiring less stringent stability bounds than the explicit schemes. Each iteration in an implicit scheme, however, involves global data dependencies in the form of second and higher order recurrences. Efficient parallel implementations of such iterative methods, therefore, are considerably more difficult and non-intuitive. In this paper, we consider the parallelization of the implicit schemes that are used for solving the Euler and the thin layer Navier-Stokes equations and that require inversions of large linear systems in the form of block tri-diagonal and/or block penta-diagonal matrices. We focus our attention on three-dimensional cases and present schemes that minimize the total execution time. We describe partitioning and scheduling schemes for alleviating the effects of the global data dependencies. An analysis of the communication and the computation aspects of these methods is presented. The effect of the boundary conditions on the parallel schemes is also discussed. The ARC-3D code, developed at NASA Ames, is used as an example application. Performance of the proposed methods is verified on the Victor multiprocessor system which is a message passing architecture develped at the IBM, T.J. Watson Research Center.

17. Key Words (Suggested by Author(s)) IMPLICIT, COMPUTATIONAL FLUID MECHANICS, PARALLELIZATION		18. Distribution Statement 64 - Numerical Analysis; Unclassified - Unlimited	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 25	22. Price A03

NASA FORM 1626 OCT 86

NASA-Langley, 1990